Capstone Project

Machine Learning Engineer Nanodegree

Paul Singman
April 25th, 2016

Developing an Airbnb Price Model

Project Overview

The goal of this project is to develop a model that predicts the listing price of a property on Airbnb. Airbnb is a service that allows people to "list, discover, and book unique accommodations around the world." The way it works is people list their homes, apartments, or spare rooms on the Airbnb site to be rented for a little as one night or as long multiple months. A common trouble people face is deciding the appropriate price for their listing, which is where my Listing Price Model enters the picture.

With a model trained on current listings, someone looking to list their property could input its relevant features, and be given a recommendation for a listing price.

The dataset for this model will be obtained by scraping the information from Airbnb's website. Listings will be collected from six major metropolitan areas: New York, Los Angeles, San Francisco, Chicago, Washington DC, and Dallas. Information about a listing --- like the No. of bedrooms, the location, and the property type --- are provided with each listing and will be considered as features for the prediction model. The listing price will be captured and used as the 'truth' we are trying to predict with our supervised learning algorithm.

Problem Statement

As stated above, the problem to be solved is determining an appropriate listing price for a property on Airbnb. The first step after collecting the dataset is to perform exploratory data mining. This entails exploring patterns in the data, finding potential outliers and/or missing values, and calculating the correlations between different features. Feature transformations such as one-hot encoding or applying numerical transformations will be considered.

Since we are trying to predict price, which is continuous, the goal will be to minimize the prediction error of the supervised regression model. Since there is no objective way to determine when predictions are close enough, the first step will be to fit a baseline model that we can hopefully improve upon. The baseline model will predict a city's average price for a listing, so for example if a listing is in New York, we'll predict the average price of all Airbnb properties in New York for that listing. If whatever future regression models we build afterwards cannot beat the error of this simple model, then it's safe to say the model is not very useful.

Metrics

The primary metric we will use to assess the effectiveness of the model is Root Mean Square Error (RMSE). As its name suggests, RMSE is calculated by taking the root of the mean of the square of the model's prediction error. The goal will be to minimize the RMSE of the model, meaning our predicted prices are as close to the actual listing price as possible, with a particular emphasis on minimizing the model's largest prediction errors. I believe this emphasis on punishing large errors is appropriate for this use case, since mispricing a property by $5 probably won't have much of an impact on rentals. Mis-pricing by $50 or $100 dollars however, would likely prevent a property from being rented at all.

In addition, the model's R-Squared statistic will be tracked. For this project, the R-Squared value will show what percent of variation in listing price is being explained by the model. An R-Squared of 0 means it is explaining none of the variation in price, and an R-Squared of 1 means it is explaining all of it.

My hope is someone looking to list their property on Airbnb will have confidence that the model gives them an appropriate listing price suggestion.

Analysis

Data Exploration

Note: Data was collected in the Jupyter Notebook called **scraper.ipynb**. Analysis of the data took place in the Notebook called **no-log-analysis.ipynb**.

The data collected for this model has the following data types and descriptions:

Bedrooms : {discrete, integer} No. of bedrooms in the property
Capacity : {discrete, integer} Ideal capacity of the property according to the lister
Review Count : {discrete, integer} No. of reviews for the listing
Room Type : {categorical, string} Either 'Entire home/apt', 'Private Room' or 'Shared Room'
Property Type : [categorical, string}
Star Rating : {discrete, float} Avg rating of property by previous renters
Price : {discrete, float} Cost to rent listing for a night

Although Price was captured as a float, after examining its values, it was found to only contain whole-number prices and could be stored as an integer. Basic summary statistics of the numeric variables are as follows from Pandas describe function:

	Bedrooms	Capacity	Price	Review_Count	Star_Rating
count	1836.000000	1836.000000	1836.000000	1836.000000	1836.000000
mean	0.934096	2.224401	88.504357	27.058279	4.064815
std	0.420863	0.978111	40.695848	42.568072	1.805952
min	0.000000	1.000000	10.000000	0.000000	0.000000
25%	1.000000	2.000000	60.000000	4.000000	4.500000
50%	1.000000	2.000000	84.000000	12.000000	5.000000
75%	1.000000	2.000000	110.000000	29.000000	5.000000
max	3.000000	12.000000	285.000000	432.000000	5.000000

As you can see, information was captured on 1836 Airbnb listings. It appears that the majority of listings have 1 Bedroom, a Capacity of 2, and a "perfect" Star Rating of 5. Looking at histograms of these variables will further elucidate how they are distributed. Before we look at those, I want to use Pandas value counts function to see the values in the Room and Property Type variables:

```
In [21]: df_all_unprocessed.Property_Type.value_counts()

Out[21]: Apartment          1154
         House               441
         Condominium          84
         Loft                 54
         Townhouse            42
         Other                18
         Bed & Breakfast      15
         Bungalow             11
         Cabin                 8
         Dorm                  4
         Villa                 2
         Castle                2
         Camper/RV             1
         dtype: int64
```

Property Type has many different values within the category, including a few like 'Castle' and 'Villa that only appear a few times. This will make it difficult to include as a feature in the ML algorithm.

Now Room Type:

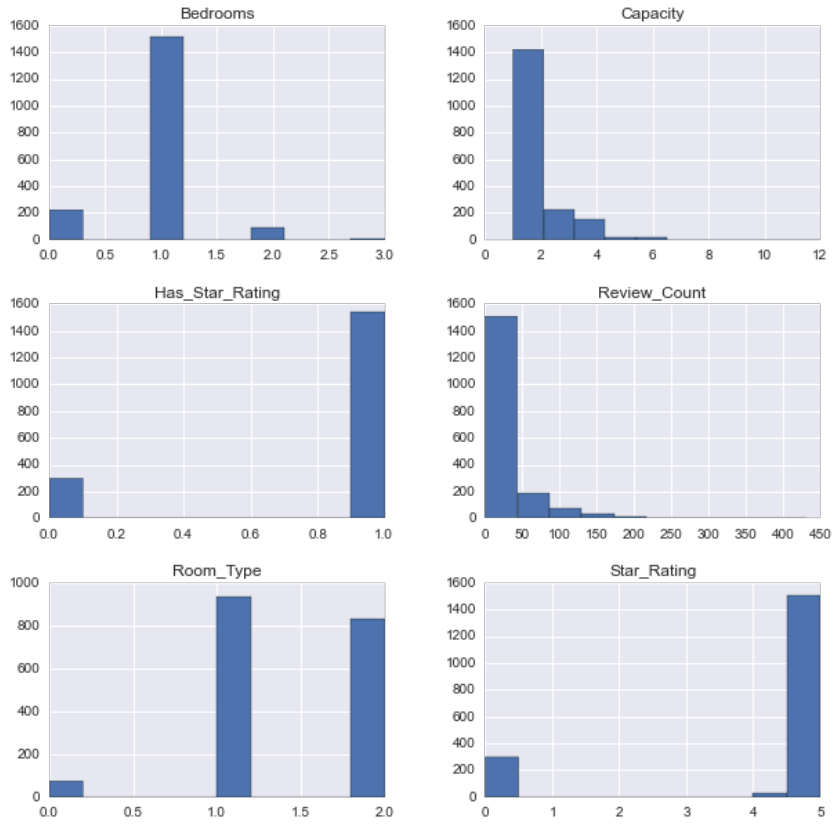
```
In [22]: df_all_unprocessed.Room_Type.value_counts()

Out[22]: Private room       934
         Entire home/apt    828
         Shared room         74
         dtype: int64
```

Room type is much more compact than Property Type and therefore will be easier to include as a feature in the model. Even further whereas Property Type is nominal (i.e. there is no intrinsic ranking to the cateogries) I believe it is appropriate to make the assumption that Room Type is ordinal (there is an intrinsic ranking), with 'Entire home/apt' the most desirable and 'Shared room' the least. With this assumption Room Type can be mapped to a numeric feature with the following relation:

Room Type mapping: {'Shared Room' : 0, 'Private Room' : 1, 'Entire home'/apt : 2}

Let's look at histograms for these variables:

Figure 1: Feature Histograms



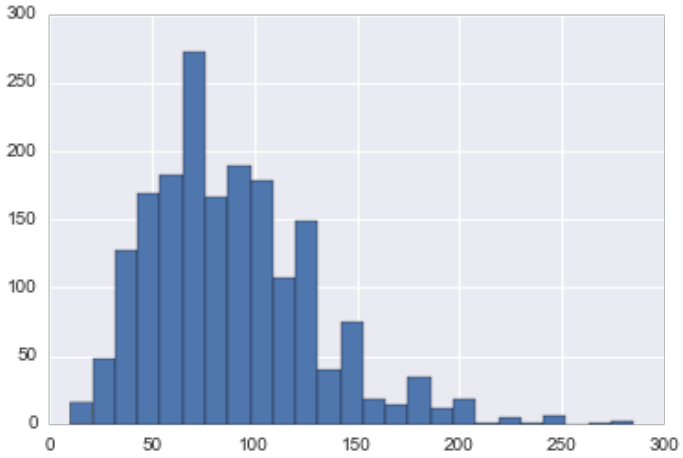
Some thoughts: Capacity and Review Count left skewed and distributed as expected. Star Rating is surprisingly concentrated around only 0 and 4.5-5. For this reason, I created a binary feature 'Has Star Rating' to capture most of the information in the variable more efficiently.

Still, I'm curious if there's a relationship between Price and Star Ratings that are not zero.

Interesting, properties with a no Star Rating have almost as high a Price (on average) as those with a perfect 5.0 Star Rating. Overall there is some trend that the lower Star Ratings of 3 and 4 do correspond to lower Prices. For this reason I'll still consider including the raw Star Rating in the model so I don't throw that information out.

Finally let's take a look at a histogram for Price:

Figure 2: Price Histogram

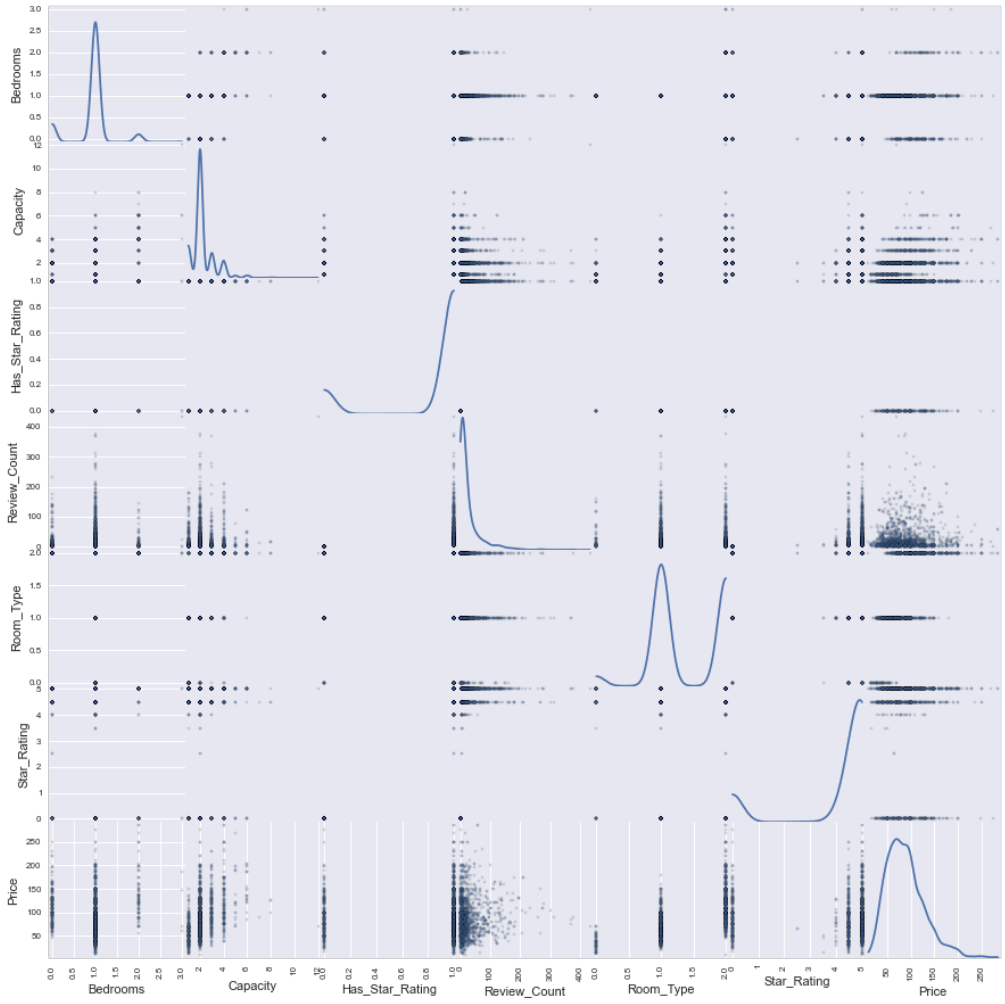


Price is relatively normally distributed, though slightly left skewed, which is not a problem.

Exploratory Visualization

Let's take a look at a scatter matrix of all of the variables, which will depict the correlation between features and the response:

Figure 3: Scatter Matrix (with Kernel Density Plots on Diagonals)



It does not seem like any of the features are particularly correlated with either each other or Price. Though we can look at a correlation matrix to view the exact correlation values:

Correlation matrix

```
In [29]: df_all[sm_cols].corr()
```

Out[29]:

	Bedrooms	Capacity	Has_Star_Rating	Review_Count	Room_Type	Star_Rating	Price
Bedrooms	1.000000	0.317923	-0.054905	0.014055	-0.162514	-0.045642	-0.010372
Capacity	0.317923	1.000000	-0.015300	0.049147	0.443049	-0.007313	0.396855
Has_Star_Rating	-0.054905	-0.015300	1.000000	0.271267	-0.030215	0.991021	-0.003583
Review_Count	0.014055	0.049147	0.271267	1.000000	-0.035037	0.273466	0.040921
Room_Type	-0.162514	0.443049	-0.030215	-0.035037	1.000000	-0.027815	0.618045
Star_Rating	-0.045642	-0.007313	0.991021	0.273466	-0.027815	1.000000	0.009269
Price	-0.010372	0.396855	-0.003583	0.040921	0.618045	0.009269	1.000000

Unsurprisingly, Bedrooms and Capacity are fairly correlated at .3179. The feature best correlated with Price is Capacity, which I assume will be the most important in the ML

model. Overall there is not high correlation between the features, so I am not worried about detrimental effects of collinearity. Also, I don't believe using a dimensionality reduction algorithm like PCA or ICA will be useful in reducing the dimension of the data while capturing a large percentage of the variance.

Algorithms and Techniques

The ML models I will fit to this dataset are variations of linear regression. Specifically I plan to fit:

1. Ordinary Least Squares – Vanilla least squares regression with no regularization
2. Ridge – Regression with L2 regularization
3. Lasso – Regression with L1 regularization
4. ElasticNet – Regression with a combination of L1 and L2 regularization

I believe with a relatively small dataset, a simpler algorithm like linear regression will have comparable performance to more complex algorithms. The benefit is there are few parameters to tune, the main one being the regularization parameter alpha. For ElasticNet there's an additional parameter to tune, the ratio of L1 and L2 regularization.

Linear regression models are also highly interpretable with the impact of each feature easily understood by the value of its corresponding coefficient. The values of the feature coefficients in a linear regression model at the end of training (along with the intercept) are the values such that the sum of squared error is minimized.

While fitting all four-regression models is a bit of overkill, my desire is to gain intuition of how regularization affects model performance and the values of the final coefficients.

Benchmark

The benchmark that I hope future models beat, is a model that uses the city's average listing price as the predicted listing price for a property. For example, the average price of all properties in New York is $92.91. For all New York properties, therefore, the predicted price will be $92.91, which can be compared to the actual listing prices to calculate the RMSE and R-Squared.

The RMSE and R-Squared of this benchmark model are 38.832 and 0.110 respectively. The percent improvement in these two metrics over the baseline model will be reported in the final model.

Methodology

Data Preprocessing

I performed preprocessing on several of the features.

- Has Star Rating – As already mentioned, I created a binary 1/0 version of Star Rating called 'Has Star Rating' due to its distribution
- Room Type – Also as already mentioned, Room Type was transformed from a categorical variable to an ordinal numeric one with the following mapping:

Room Type mapping: {'Shared Room' : 0, 'Private Room' : 1, 'Entire home'/apt : 2}

- City – One-hot encoded the city feature, transforming it into 5 features as shown below:

Figure 4: One-hot encoding example for listings in NY

City_DA	City_DC	City_LA	City_NY	City_SF
0	0	0	1	0
0	0	0	1	0
0	0	0	1	0
0	0	0	1	0
0	0	0	1	0

Linear regression does not require its features to be standardized or scaled, so no additional feature transformations are required to apply the model to the datset.

Implementation

The Scikit Learn Machine Learning package will be used to train and create predictions with the linear regression models.

The first step will be splitting the data into training and testing sets with the train_test_split function:

```
In [53]: regression_features = ['Star_Rating','Has_Star_Rating','Bedrooms','Room_Type', 'Capacity','Review_Count',
                                'City_NY','City_SF','City_DA','City_DC','City_LA']
         X = df_all[regression_features]
         y = df_all.Price

         # split the data into training and testing sets
         X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 123)
```

Next we instantiate instances of the appropriate Scikit classes (with default parameters to start):

```
In [45]: lr_ridge = Ridge()
         lr_ols = LinearRegression()
         lr_lasso = Lasso()
         lr_elasticnet = ElasticNet()
```

Next the process of training the model on the training data, using the trained model to generate price predictions on the test data, and scoring the predictions to the actual price is carried out in four functions.

The first function called 'regression_model' takes in the instance of the classifier as well as the training and testing data. It calls the subsequent functions that complete the other steps of the process and returns the trained instance of the classifier. The other three functions called 'train_model', 'model_predictions', and 'score_model' are self-explanatory. 'train_model' prints out the training time of the classifier, and 'regression_model' prints the R-Squared on the test set and the RMSE on the training and test sets.

The code for all four functions is copied below, in order that they are executed in the notebook:

In [11]:
```
def regression_model(clf, X_train, X_test, y_train, y_test):
    """This function:
     - trains the model on the training
     - makes predictions on the test set
     - reports the r_squared and RMSE of the model
     - returns the instance of the classifier
    """
    clf = train_model(clf, X_train, y_train)
    train_predictions, test_predictions = model_predictions(clf, X_train, X_test)
    train_RMSE, test_RMSE, test_rsquare = score_model(train_predictions, test_predictions, y_train, y_test)

    print "RMSE for training set: {}".format(train_RMSE)
    print "RMSE score for test set: {}".format(test_RMSE)
    print "R-Squared for test set: {}".format(test_rsquare)

    return clf
```

In [8]:
```
def train_model(clf, X_train, y_train):
    """Trains the classifier"""
    print "-----------------------------"
    print "Training {}...".format(clf.__class__.__name__)
    start = time.time()
    clf.fit(X_train, y_train)
    end = time.time()
    print "Done!\nTraining time (secs): {:.3f}".format(end - start)

    return clf
```

In [9]:
```
def model_predictions(clf, X_train, X_test):
    """Generate predictions from the training and testing data"""
    train_predictions = clf.predict(X_train)
    test_predictions = clf.predict(X_test)

    return (train_predictions, test_predictions)
```

In [10]:
```
def score_model(train_preds, test_preds, train_truth, test_truth):
    """Return the R-squared on the test data, and the RMSE on the train and test data"""
    train_RMSE = np.sqrt(metrics.mean_squared_error(train_truth, train_preds))
    test_RMSE = np.sqrt(metrics.mean_squared_error(test_truth, test_preds))

    test_rsquare = metrics.r2_score(test_truth, test_preds)

    return (train_RMSE, test_RMSE, test_rsquare)
```

In order to go through the process outlined above, one simply needs to call the 'regression_model' function. We'll start with the Ordinary Least Squares model:

```
In [48]: lr_ols = regression_model(lr_ols, X_train, X_test, y_train, y_test)

----------------------------
Training LinearRegression...
Done!
Training time (secs): 0.001
RMSE for training set: 27.3323151303
RMSE score for test set: 28.3666468061
R-Squared for test set: 0.522236842795
```

We can see that these results dramatically outperform the baseline model fit earlier. The test RMSE for the baseline model was 38.832 and the R-Squared was 0.11. With the OLS model the test RMSE has dropped to 28.36 and the R-Squared increased to 0.522.

We knew the regression model would outperform the baseline though; let's see how OLS compares to the other regression models, which contain regularization parameters. We'll start with Ridge:

```
In [46]: lr_ridge = regression_model(lr_ridge, X_train, X_test, y_train, y_test)

----------------------------
Training Ridge...
Done!
Training time (secs): 0.002
RMSE for training set: 27.3472326159
RMSE score for test set: 28.3380972765
R-Squared for test set: 0.523198045545
```

The RMSE on the test set for the Ridge model slightly outperforms the OLS model, 28.338 vs 28.366. Let's quickly see how the Lasso and ElasticNet models compare:

```
In [52]: lr_lasso = regression_model(lr_lasso, X_train, X_test, y_train, y_test)

----------------------------
Training Lasso...
Done!
Training time (secs): 0.002
RMSE for training set: 28.3009203644
RMSE score for test set: 28.9302308841
R-Squared for test set: 0.503064008226
```

```
In [50]: lr_elasticnet = regression_model(lr_elasticnet, X_train, X_test, y_train, y_test)

----------------------------
Training ElasticNet...
Done!
Training time (secs): 0.002
RMSE for training set: 33.3207798611
RMSE score for test set: 33.2261363697
R-Squared for test set: 0.34452468668
```

We see that the Lasso regression model has similar performance to OLS and ridge, but the ElasticNet model lags behind with an RMSE of 33.226. I'd like to optimize the alpha and l1_ratio parameters of the ElasticNet model but first, let's take a look at the feature coefficients of all four models.

My expectation is that OLS will have the largest coefficients and Lasso will have the smallest, since the effect of regularization is to shrink the coefficients to prevent over-fitting the dataset while training:

Learned Coefficients for Regression Models				
Features	OLS	Ridge	ElasticNet	Lasso
Star Rating	16.1	12.5	0.2	0.2
Has Star Rating	-76.31	-59.0	-0.0	0.0
Bedrooms	6.1	6.0	-0.2	0.0
Room Type	42.0	41.9	14.4	37.8
Capacity	6.5	6.6	8.0	6.5
Review Count	0.01	0.01	0.03	0.03
City_NY	23.7	23.0	1.3	9.1
City_SF	41.9	41.3	6.4	28.1
City_DA	-0.99	-1.8	-1.9	-0.7
City_DC	10.3	9.7	0.0	0.0
City_LA	6.4	5.9	-1.5	0.0

Looking at the coefficient values, we see the effect of regularization is apparent. Lasso regression has the 'sparsest' solution, meaning the largest of coefficients with a value of 0. Interestingly, the value of its coefficients that are not zero are larger than ElasticNet's, which is not a result I was aware beforehand. The ElasticNet model has some interesting properties with some of the sparsity of the Lasso model combined with the most extreme coefficient shrinkage.

Let's refine the ElasticNet model!

Refinement

The alpha (regularization) and l1_ratio parameters will be tested over the following values to determine the optimal values:

```
In [28]: l1_ratios = [.05,.1, .5, .7, .9, .95, .99, 1]
         alphas = [10e-4, 10e-3, 10e-2, 10e-1, 0, 1, 10, 100, 1000]
```

Then the following code will be used to loop through all combinations of these parameters:

```
In [31]: for r in l1_ratios:
             for a in alphas:
                 lr_elasticnet = ElasticNet(alpha=a, l1_ratio=r)
                 print "l1 ratio: {}, alpha: {}".format(r, a)
                 lr_elasticnet = regression_model(lr_elasticnet, X_train, X_test, y_train, y_test)
                 print"\n"
```

The results of each trial can be found in the accompanying Jupyter Notebook. The result, though, is that an ElasticNet model with an alpha of 0.01 and an l1_ratio of 0.1 has the best performance on the test set. It's results are:

```
In [55]: lr_elasticnet = regression_model(lr_elasticnet, X_train, X_test, y_train, y_test)

         ----------------------------
         Training ElasticNet...
         Done!
         Training time (secs): 0.005
         RMSE for training set: 27.5639573115
         RMSE score for test set: 28.319962361
         R-Squared for test set: 0.523808107426
```

The RMSE on the test data of 28.3199 greatly outperforms the 33.32 mark from the un-optimized ElasticNet model, and even outperforms the previous best Ridge model.

Let's compare the coefficient values between the original and optimized ElasticNet models:

Learned Coefficients for Regression Models		
Features	ElasticNet (orig)	ElasticNet (optimal)
Star Rating	0.2	3.85
Has Star Rating	-0.0	-16.0
Bedrooms	-0.2	5.1
Room Type	14.4	40.4
Capacity	8.0	7.0
Review Count	0.03	0.02
City_NY	1.3	18.6
City_SF	6.4	36.3
City_DA	-1.9	-4.1
City_DC	0.0	6.0
City_LA	-1.5	2.4

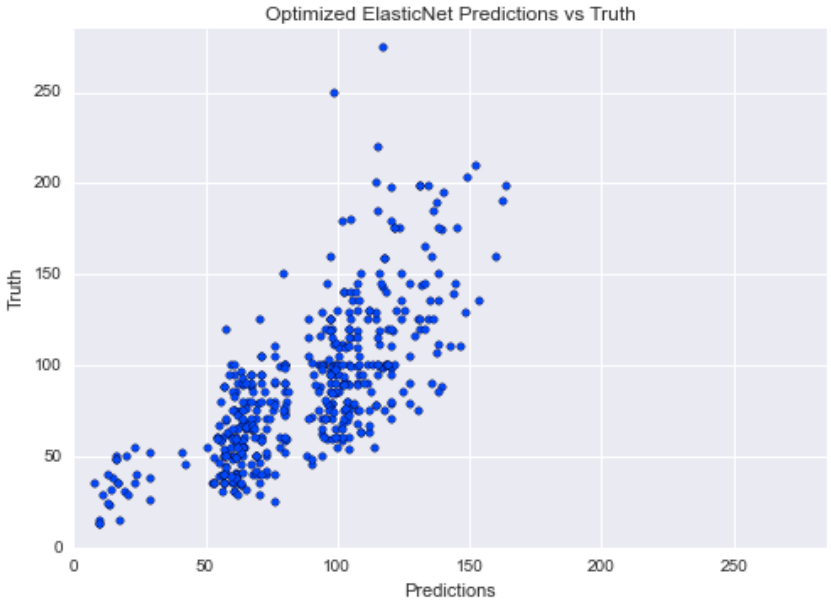
Interestingly, the optimized ElasticNet model reduced some of the sparsity of the original, with non-zero values for the 'Has Star Rating' and 'City_DC' features. The values of other features like 'Star Rating', 'Room Type', and 'City NY' saw their coefficients increase however, closer to the midpoint of the Ridge and Lasso regression models.

Results

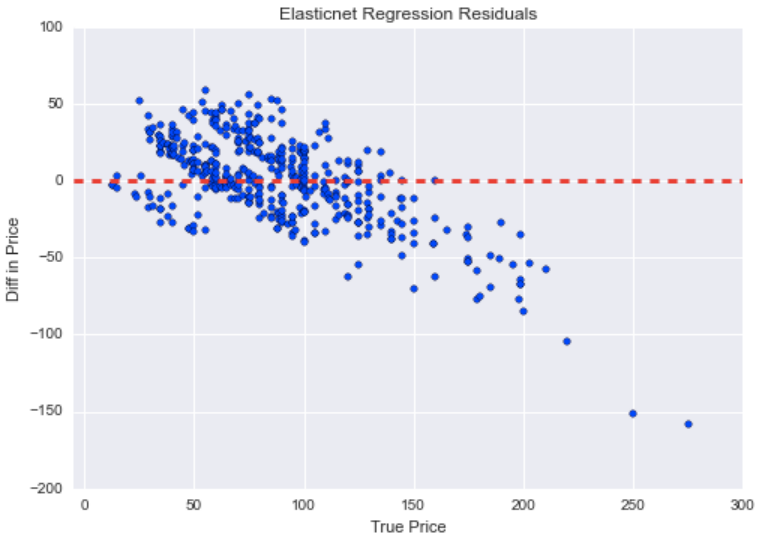
Model Evaluation and Validation

The final model is an ElasticNet regression model, with an alpha of 0.01 and an l1_ratio of 0.1. I thought with my relatively small dataset, a linear regression model would perform adequately for this task. While all of the variations of regression that I tested (OLS, Ridge, Lasso, and ElasticNet) performed similarly, I find the final coefficient values for the optimized ElasticNet model most intuitive.

To evaluate the model let's take a look first at it's predicted vs true values, and then a graph of the residuals.



As you can see, the model does not perform perfectly on the dataset, particularly it appears to under-predict the price of the most expensive listings. Let's see if the residuals plot confirms this, ordered form least to most expensive:



Yes, the residuals plot confirms that the most expensive properties are being under-predicted. The model does, however, perform reasonably well in the $25-$150 range.

Let's look at the feature values of some of the expensive listings:

```
In [60]: high_price_df.head()
```

Out[60]:

	Bedrooms	Capacity	Price	Review_Count	Room_Type	Star_Rating
3	1	4	250	9	2	4.5
13	1	2	275	1	2	0.0
3	0	2	250	7	2	5.0
11	2	4	285	56	2	5.0
11	1	2	250	3	2	4.5

Looking at the feature values, it is not particularly surprising that these properties are being underpredicted. Having a higher No. of Bedrooms or large Capacity is the most straightforward way for the model to predict an expensive Price, and as you can see, none of these listings have a particularly large No. of Bedrooms or Capacity.

There must be other factors like neighborhood or luxurious furnishings in the apartments that are causing the high price, and the data collected that feed into the model does not capture this information.

Despite this shortcoming, since the model was trained on a training dataset and tested and scored on unseen data points, we can be confident that the model will generalize well for the types of properties it already performs well on.

Justification

Compared to the baseline, the final model performs extraordinarily better. The final model's RMSE is 37% lower, and the R-Squared is 376% higher.

I believe with the data available, the final model's performance is as well as one could expect, particularly outside of the luxury market. However, I would not say my analysis and model "solves the problem", and in the next section I will go into specifics of how it could be expanded upon and improved.

Conclusion

Free-Form Visualization

A final visualization I'd like to share for this project looks like this:

It shows how performance (in terms of RMSE) changes over different training set sizes. Two interesting takeaways are:

1. The test data performance does not improve after the training set is larger than 200.
2. The gap between the performance on the training and test narrows considerably when the training set size is larger than 800. This suggests that gathering additional data will not significantly improve the performance of the classifier.

Reflection

I felt this project was a great way to demonstrate and build my skills across the entire data science workflow. In scraping the Airbnb website, I demonstrated the ability to acquire data "in the wild". Part of the challenge in collecting my own dataset was determining what data to collect and determining what features would be most useful in a ML model.

Numerous features required transformations like one-hot encoding the 'City' feature and mapping the categorical feature 'Room Type' to a numeric one. Then, my favorite part of the project was fitting the different regression models, to gain a better intuitive sense of how regularization affects the feature coefficients. The results were both partly what I expected (regularization shrinks parameter values), and partly unexpected (in how the size of the coefficient varies between L2 and L1 regularization).

Improvement

The model performs reasonably well for mid-priced Airbnb properties, however, it does lack the ability to make reasonable predictions for some high-end listings. Taking into account neighborhood location or creating a feature based on NLP of the description of a property are two potential ways to improve the model for these high-end listings in particular.

There is potential to fit more complex ML algorithms on this dataset to improve the performance. A Gradient Boosted Regression Tree could potentially perform better, but I would not feel comfortable implementing that algorithm without collecting significantly more data points.

If I were to expand this into a longer term project, I would scrape the Airbnb site over time, allowing for the inclusion of seasonality effects into the model. So room for improvement certainly exists, though I feel my Airbnb Listing Price Prediction model is capturing significant information on what determines a listing's price.